StarPU-MPI: Task Programming over Clusters of Machines Enhanced with Accelerators

Cédric Augonnet, Olivier Aumage, Nathalie Furmento, Samuel Thibault, Raymond Namyst

StarPU-MPI: Task Programming over Clusters of Machines Enhanced with Accelerators

Cédric Augonnet, Olivier Aumage, Nathalie Furmento, Raymond Namyst, Samuel Thibault

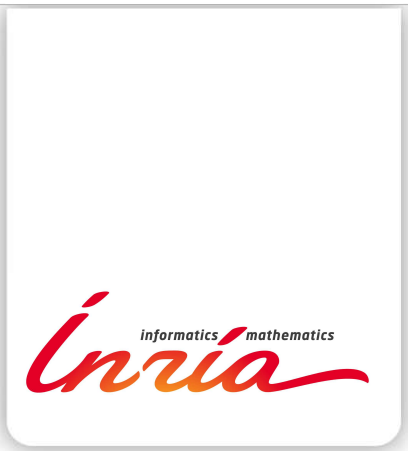
RESEARCH REPORT

N° 8538

May 2014

Project-Teams Runtime

ISSN 0249-6399 ISRN INRIA/RR--8538--FR+ENG



StarPU-MPI: Task Programming over Clusters of Machines Enhanced with Accelerators

Cédric Augonnet, Olivier Aumage, Nathalie Furmento,
Raymond Namyst, Samuel Thibault

Project-Teams Runtime

Research Report n° 8538 — May 2014 — 12 pages

Abstract: GPUs have largely entered HPC clusters, as shown by the top entries of the latest top500 issue. Exploiting such machines is however very challenging, not only because of combining two separate paradigms, MPI and CUDA or OpenCL, but also because nodes are heterogeneous and thus require careful load balancing within nodes themselves. The current paradigms are usually limited to only offloading parts of the computation and leaving CPUs idle, or they require static work partitioning between CPUs and GPUs.
To handle single-node architecture heterogeneity, we have previously proposed StarPU, a runtime system capable of dynamically scheduling tasks in an optimized way on such machines. We show here how the task paradigm of StarPU has been combined with MPI communications, and how we extended the task paradigm itself to allow mapping the task graph on MPI clusters such as to automatically achieve an optimized distributed execution. We show how a sequential-like Cholesky source code can be easily extended into a scalable distributed parallel execution, and already exhibits a speedup of 5 on 6 nodes.

Key-words: Accelerators, GPUs, MPI, Task-based model

RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

StarPU-MPI: Paradigme de Tâches sur des Grappes Hybrides de Calcul

Résumé : Les GPUs ont une place importante en HPC comme le montre les dernières listes du top500. Exploiter de telles machines est toutefois difficile, du fait de l'utilisation de deux paradigmes, MPI et CUDA ou OpenCL, mais aussi du fait de l'hétérogénéité des noeuds. Trouver un bon équilibrage de charge est de ce fait primordial. Les paradigmes actuels se limitent normalement à déporter des bouts de calcul en laissant les CPUs inactifs, ou bien requièrent un partitionnement statique entre les CPUs et les GPUs.

StarPU, un support d'exécution permettant d'ordonnancer dynamiquement et de manière optimisée des tâches, sait déjà tirer parti au mieux de l'hétérogénéité d'une architecture avec un seul noeud. Nous montrons dans ce papier comment le paradigme de tâches de StarPU a été combiné avec les communications MPI, afin de permettre un placement d'un graphe de tâches sur une grappe MPI, et ce pour obtenir de manière automatique une exécution distribuée optimale. Nous montrons comment un code séquentiel Cholesky peut facilement être adapté pour une exécution distribuée passant à l'échelle, et montre déjà une accélération sur 5 ou 6 noeuds.

Mots-clés : Accélérateurs, GPUs, MPI, Modèle de tâches

StarPU-MPI: Task Programming over Clusters of Machines Enhanced with Accelerators

Cédric Augonnet[1], Olivier Aumage[3], Nathalie Furmento[2], Raymond Namyst[2], and Samuel Thibault[2]

[1] NVIDIA Corporation, Santa Clara, California, USA
[2] LaBRI, CNRS, University of Bordeaux, France
[3] Inria, Bordeaux, France

GPUs have largely entered HPC clusters, as shown by the top entries of the latest top500 issue. Exploiting such machines is however very challenging, not only because of combining two separate paradigms, MPI and CUDA or OpenCL, but also because nodes are heterogeneous and thus require careful load balancing within nodes themselves. The current paradigms are usually limited to only offloading parts of the computation and leaving CPUs idle, or they require static work partitioning between CPUs and GPUs.

To handle single-node architecture heterogeneity, we have previously proposed StarPU, a runtime system capable of dynamically scheduling tasks in an optimized way on such machines. We show here how the task paradigm of StarPU has been combined with MPI communications, and how we extended the task paradigm itself to allow mapping the task graph on MPI clusters such as to automatically achieve an optimized distributed execution. We show how a sequential-like Cholesky source code can be easily extended into a scalable distributed parallel execution, and already exhibits a speedup of 5 on 6 nodes.

Keywords: Accelerators, GPUs, MPI, Task-based model

1 Introduction

Problem sizes encountered in many HPC applications make the simultaneous use of multiple machines necessary. GPU have now fully entered the HPC arena, as shown by the top entries of the latest top500 issue. A lot of research has thus been conducted to allow MPI applications to offload code on GPU devices [1, 2]. However, solutions often only make it easier to offload some parts of the computation to GPUs, and leave the CPUs idle. Or they make it easier to run computations on all CPUs and GPUs of the cluster concurrently, but they require some predetermined distribution of work between CPUs and GPUs. Such approaches are not portable since the distribution depends both on the application kernel efficiency and the actual heterogeneous hardware relative speed.

StarPU is a runtime system capable of scheduling tasks over one multicore machine equipped with accelerators such as GPUs. It uses a software virtual shared memory (VSM) that provides a high-level programming interface and automates data transfers between processing units so as to enable a dynamic scheduling of tasks over all the hybrid processing units.

The integration of StarPU and MPI can take different aspects, depending on whether we accelerate existing MPI codes, or whether we distribute existing StarPU applications over clusters.

Firstly, provided the huge amount of MPI applications, we need to make it possible to accelerate existing MPI applications so that they can take full advantage of accelerators thanks to StarPU. Our approach is to have an instance of StarPU initialized on each MPI node. The flexibility of such an hybrid programming model has already been illustrated in the case of MPI applications which call libraries written in OpenMP or TBB for instance. It is then up to the application to decide which MPI node should submit a given task to its local instance of StarPU. Now, even though data management can be performed by StarPU within an MPI node, a message-passing paradigm implies that the different nodes should be able to exchange data managed locally by StarPU. This approach is presented in Section 2.

Secondly, we must provide a convenient way to write new applications, or to extend StarPU applications, or more generally task-based applications, so that they can exploit clusters. There is a real opportunity for applications, that are not already too large or too complex, to be rewritten using a task-based paradigm. The task paradigm provides a powerful abstraction of algorithms that can be efficiently mapped on very different types of platforms, going from single-node multicore machines to large clusters of multicore machines equipped with accelerators. In Section 3, we explain how we extended the task paradigm in StarPU so that applications can fully exploit clusters of multicore machines enhanced with accelerators. By combining StarPU with MPI, we gain full benefit from both paradigms: scheduling tasks over CPUs and GPUs, and using clusters equipped with GPU devices. Section 4 shows the performance we are getting with this new approach. We present related works in Section 5 before concluding.

2 Integrating MPI Communications in StarPU

To illustrate how to integrate MPI communications within StarPU, we start from a trivial example which simply passes a token between machines. For each loop, each node receives the value from its predecessor, performs its own computation, and sends the resulting value to its successor.

```
1 for (loop = 0; loop < nloops; loop++) {
2   if (!(loop == 0 && rank == 0))
3     MPI_Recv(&token, 1, MPI_UNSIGNED, (rank+size-1)%size, 0, MPI_COMM_WORLD, NULL);
4   increment_token_cpu(&token);
5   if (!(loop == last_loop && rank == last_rank))
6     MPI_Send(&token, 1, MPI_UNSIGNED, (rank+1)%size, 0, MPI_COMM_WORLD);
7 }
```

Now, we would like to be able to offload the computation on GPU devices. Doing this *by hand* would be rather tricky as one would need to take care of the data transfers between the main memory and the GPU memory within a node, and to ensure synchronization with the MPI data transfers between nodes. Instead, StarPU can takes care of all these management operations and let programmers focus on their application. For that, StarPU has been extended

with a library that provides an MPI-like semantic to ease the integration of StarPU into existing codes and to easily parallelize StarPU applications with MPI.

This library, a preliminary version of which has been presented in [3], is implemented on top of a StarPU low-level facility, the non-blocking acquire/release semantic which provides a powerful mechanism which can be used to exchange registered data between different processes. The goal of this library is to provide an effective interface to implement message passing semantics between multiple instances of StarPU, in a consistent way with StarPU's data management infrastructure. For the programmers, the change is minimal. They simply specifies StarPU's data handles instead of data pointers and lengths when performing message passing transfer operations. The conversion between StarPU's data handles and actual data pointers and lengths is then performed transparently within the library.

Following on the ring example, the source code below illustrates how simple it is to transfer data between multiple instances of StarPU by the means of our MPI-like library. The detached calls used in the example perform *asynchronous* data transfers. Thanks to an internal communication progress mechanism implemented within the library, the data handle is then automatically released when the completion of the data transfer is detected. MPI communications are all performed by a single dedicated StarPU thread.

```
1  starpu_vector_data_register(&token_handle, 0, &token, 1, sizeof(unsigned));
2  for (loop = 0; loop < nloops; loop++) {
3    if (!(loop == 0 && rank == 0))
4      starpu_mpi_irecv_detached(token_handle, (rank+size-1)%size,
5                                0, MPI_COMM_WORLD, NULL, NULL);
6    starpu_insert_task(&increment_token_cl, STARPU_RW, token_handle, 0);
7    if (!(loop == last_loop && rank == last_rank))
8      starpu_mpi_isend_detached(token_handle, (rank+1)%size,
9                                0, MPI_COMM_WORLD, NULL, NULL);
10 }
11 starpu_task_wait_for_all();
```

As in the MPI code, each node gets the current value from its predecessor (line 4), performs local computations using a StarPU task (line 6), and sends the updated value to its successor (line 8). The absence of explicit dependencies between these different operations shows that our extension is nicely integrated with implicit dependencies which not only apply to tasks but also to other types of data accesses, here the acquisition of data for the MPI transfer.

To summarize, the for loop merely submits all MPI communications requests and tasks with implicit dependencies to StarPU, without blocking, and the last line waits for StarPU to complete them all in an optimized way. It is worth noting that even though the data was registered by every MPI node (line 2), the coherency of these data is managed independently by the different instances of StarPU. Indeed, instead of implementing a distributed shared memory that would manage handles in a distributed fashion, our MPI-like library provides a convenient API to transfer the content of a handle into another handle which was registered by another node.

3 Adapting the StarPU Task-Based Paradigm to a Cluster Environment

One of the motivations for adopting a task-based model is that describing an application as a graph of tasks is generic enough to allow a runtime system to map the graph on a variety of platforms. Task graphs are indeed a convenient and portable representation which is not only suited to hybrid accelerator-based machines, but also to clusters of nodes enhanced with accelerators.

3.1 Towards Task-Based Programming

Since StarPU can automatically infer data dependencies from task submission itself, the source code below, although very close to the typical sequential implementation for the Cholesky decomposition, is actually a parallel code: it submits all the tasks to StarPU in the sequential order, and these are turned into a task graph thanks to the inferred data dependencies. By using macros, it is actually very simple to use the same source for both sequential and StarPU versions. This has been previously generalized into a hybridization methodology for linear algebra software [4]. It was implemented within the MAGMA dense linear algebra library, which from its version 1.1 uses StarPU to dynamically schedule its tasks among CPUs and GPUs. It permitted to work seamlessly on algorithmic improvements on Cholesky [5], LU [6], and QR [7] for example. We now show that this methodology can actually be extended to provide a similar semantic in a distributed environment.

```
1  for (k = 0; k < Nt; k++) {
2    starpu_insert_task(&potrf, RW, A[k][k], 0);
3    for (m = k+1; m < Nt; m++)
4      starpu_insert_task(&trsm, R, A[k][k], RW, A[m][k], 0);
5    for (m = k+1; m < Nt; m++) {
6      for (n = k+1; n < m; n++)
7        starpu_insert_task(&gemm, R, A[m][k], R, A[n][k], RW, A[m][n], 0);
8      starpu_insert_task(&syrk, R, A[m][k], RW, A[m][m], 0);
9    }
10 }
11 starpu_task_wait_for_all();
```

3.2 A Methodology to Map DAGs of Tasks on Clusters

Figure 1 illustrates how to map a DAG of tasks on a two-node cluster. One instance of StarPU is launched on each MPI node. The first step consists in partitioning the graph into multiple sub-graphs which correspond to the tasks that will be executed by the different instances of StarPU. Assuming data-driven dependencies, dependencies that cross the boundary between two nodes are fulfilled by replacing them with a corresponding data transfer that is performed by the means of our MPI-like library. In other words, a node that generates a piece of data required by its neighbour(s) makes a send call. Similarly, a node that needs a piece of data that was generated on another node makes a receive call. Intra-node dependencies are directly managed by the instance of StarPU that schedules tasks on the corresponding MPI node. Provided an initial partitioning of the DAG, this shows that our task-based paradigm is also suited for clusters

of multicore nodes enhanced with accelerators. Our approach also leads to a task distribution which is based on the data distribution.

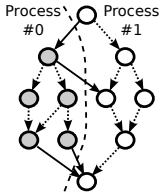


Fig. 1. Example of task DAG divided in two nodes: MPI communications are shown in full arrows, intra-node data dependencies are shown in dotted arrows.

3.3 The starpu_mpi_insert_task Helper

The first step of our method was to partition the DAG into multiple parts. Our approach to facilitate this step is to let the application define an *owner* for each data registered to StarPU: a task that modifies this handle will by default be executed on the node which owns that data. Task partitioning therefore directly results from an initial data mapping. Interestingly, this approach is commonly used to dispatch computation in a distributed environment such as MPI. A classic strategy to implement dense linear algebra kernel over MPI (*e.g.* ScaLAPACK [8]) for instance often consists in determining an initial data partitioning (*e.g.* 2D cyclic mapping of the blocks). Sequential algorithms initially designed for shared memory (*e.g.* LAPACK [9]) are then ported to a distributed environment by following this data mapping, and exchanging the various data contributions produced by the different MPI nodes during the algorithm.

The application thus first has to assign an owner node to each data handle. All MPI nodes then submit the entire DAG using the `starpu_mpi_insert_task` function, *i.e.* each node submits a task for each node in the DAG in the sequential order. A task will however actually be executed on a single node only: by default the node that owns the data that the task modifies. If the task modifies datas owned by multiple nodes, the default algorithm selects the node in a way to minimize data transfers. The other nodes do not execute the task but they automatically send a valid copy of the needed data, while the execution node automatically receives the needed data. Since every MPI node unrolls the entire DAG, StarPU actually keeps track of which MPI node already holds a valid copy of the data, and can thus avoid sending/receiving the same piece of data multiple times unless it has been modified by some task in-between.

The code below provides an MPI version of the Cholesky decomposition. The main difference with the single-node version is the initialization part. The first step consists in registering the different tiles of the matrix (line 2). The `starpu_data_set_rank` function then specifies which MPI node owns which data handle,

i.e. which node will execute the tasks that modify this matrix tile (line 3). Here, a 2D-block-cyclic distribution is used: the 2D grid is divided into blocks of size `X_BLK*Y_BLK`, and tiles within each block are distributed among nodes. The second step is to asynchronously submit all the tasks of the DAG, in sequential order (lines 5-14). The only difference with the single-node version is that `starpu_mpi_insert_task` additionally takes the MPI communicator in which transfers must be performed. Finally, the barrier on line 16 ensures that all tasks have been executed within the local MPI node. An additional optimization step not shown here is to only allocate tiles that a given node will need to work with. It should be noted that the entire DAG is unrolled regardless of the underlying data allocation and mapping. Such a separation of concerns between having a suitable data mapping and describing the application as a task graph enhances both productivity and portability.

```
1  for(x = 0; x < X; x++) for(y = 0; y < Y; y++) {
2    starpu_matrix_data_register(&A[x][y], 0, &A_tile[x][y], ld, tile_s,tile_s);
3    starpu_data_set_rank(A[x][y], (y%Y_BLK)*X_BLK + (x%X_BLK));
4  }
5  for (k = 0; k < Nt; k++) {
6    starpu_mpi_insert_task(MPI_COMM_WORLD, &potrf, RW, A[k][k], 0);
7    for (m = k+1; m < Nt; m++)
8      starpu_mpi_insert_task(MPI_COMM_WORLD, &trsm, R, A[k][k], RW, A[m][k], 0);
9    for (m = k+1; m < Nt; m++) {
10     for (n = k+1; n < m; n++)
11       starpu_mpi_insert_task(MPI_COMM_WORLD, &gemm,
12                              R, A[m][k], R, A[n][k], RW, A[m][n], 0);
13     starpu_mpi_insert_task(MPI_COMM_WORLD, &syrk, R, A[m][k], RW, A[m][m], 0);
14   }
15 }
16 starpu_task_wait_for_all();
```

3.4 Implementation Overview

Each MPI node unrolls the entire DAG described by the means of task insertion facility. Since all tasks are visible to each MPI node, and since the order of task submission is the same for all nodes, there is no need for control messages between the different MPI nodes (which behave in a deterministic way).

When a task modifies a piece of data managed by the local MPI node, task insertion results in the submission of an actual StarPU task in the local MPI node. On the one hand, if the data owner detects that another MPI node needs a valid data replicate, the data owner issues an asynchronous MPI send operation. On the other hand, the MPI node which actually needs a piece of data which is not managed by the local node issues an MPI receive operation before executing tasks that access this data replicate. Each node keeps track of the validity of its local data replicates. The MPI node which is the owner of a piece of data is always guaranteed to have a valid copy. Other nodes receive a copy of the data replicate from the owner. In order to avoid transferring the same piece of data multiple times, data replicates are kept as valid until the data owner modifies this piece of data.

There are still corner cases that need to be solved because the mapping of the tasks is directly derived from that of the data. A specific decision must for instance be taken for tasks modifying several pieces of data that are owned by different nodes. A choice must also be made for a task that does not modify

any data (*i.e.* which only has side effects). In such cases, we can break the tie using additional rules. Tasks modifying multiple data can be assigned to the owner of the first data that is modified, and results are immediately sent back to the owners of other data. Tasks which do not modify data can for instance be assigned in a deterministic fashion by the means of a round robin policy. An application may also explicitly require that a task should be scheduled by a specific MPI node.

3.5 Scalability Concerns

"MPI-zing" StarPU applications that were written using the `starpu_insert_task` function is therefore straightforward. Registering all data and having the entire DAG unrolled by each MPI node however introduces a potential scalability pitfall. We first need to ensure that the overhead is very limited when dealing with tasks that are not executed locally and which do not introduce any data transfer. Expert programmers can also be helpful by locally pruning parts of the DAG by hand. In case the programmer can determine the exact subset of data handles that will be accessed by a MPI node, there is no need to unroll the parts of the DAG whose related data handles are not accessed by this MPI node. Such pruning is also sometimes doable from high-level tools that can take advantage of a phase of static analysis.

4 Performance Evaluation

Figure 2 shows the strong scalability obtained by the Cholesky decomposition on a cluster of machines enhanced with accelerators. Each machine has two Intel Nehalem X5650 sockets with 6 cores each, running at 2.67 GHz, as well as 3 NVIDIA Fermi M2070 GPUs each. We use the **heft** StarPU scheduling policy which obtains the best single-node performance. Even though the programming effort to implement the distributed version out of the single-node version consists in a few lines of code change, our distributed Cholesky decomposition reaches almost **7 TFlop/s** (in single precision) on this **6-node** cluster with **3 GPUs and 12 CPU cores per node**. Such a speedup of approximately 5 on 6 MPI nodes is reasonable considering that the network is a serious bottleneck. It will be improved by extending StarPU's MPI-like library to support the features added in recent releases of the CUDA driver: the direct transfer capabilities introduced between CUDA devices and network cards will significantly reduce the bandwidth consumption, without having to modify the application.

5 Related Works

Various attempts have been made to provide combined MPI/GPU paradigms.

GMH [10] introduces explicit communication between all GPUs of the cluster: MPI nodes are the GPUs themselves. This approach however does not permit to use CPUs for the computation.

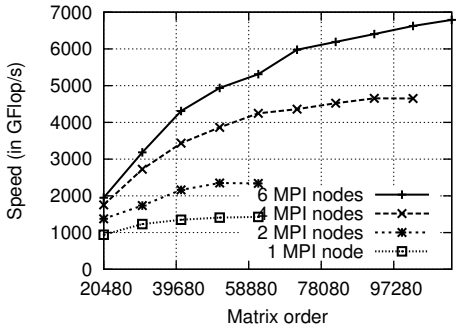


Fig. 2. Strong scalability of a Cholesky decomposition over a cluster of machines accelerated with 3 GPUs and 12 CPU cores per MPI node.

Projects such as MGP [11] or rCUDA propose OpenCL or CUDA implementations which expose GPUs on remote nodes. This permits to easily deploy existing applications over a cluster, but suffers from a centralized structure, and does not permit to exploit CPUs.

PGAS languages such as UPC or XcalableMP provide distributed shared memory and are being extended to CUDA devices [12, 13]. The proposed interfaces are however mostly guided by the application, and do not target dynamic scheduling. Even the Cholesky decomposition would be hard to schedule over all CPUs and GPUs. We are currently working with the XcalableMP team to try to extend it to use StarPU for dynamic task execution.

More generally, the StarPU data model is relatively close to Distributed Shared Memory (DSM) works. The difference is that the granularity is decided by the application (registered data), and which code parts works on which data is known to the runtime, since that is exactly what tasks express. That permits to overcome the usual shortcomings of DSMs.

The StarSs language family is currently converging into the OmpSs [14] interface, which should be able to support clusters of hybrid machines in a way very similar to what is proposed here. OmpSs master-worker scheduling policies are however still basic and do not take full advantage of heterogeneous architectures. OmpSs also imposes using a dedicated compiler to express tasks, which makes integration in existing applications harder, and the software is not publicly available yet.

The FLAME project provides the `libflame` library together with the SuperMatrix runtime. The library and the runtime cooperates to implement and schedule linear algebra algorithms on a single heterogeneous node equipped with multiple GPUs [15, 16]. Several basic scheduling algorithms are provided: single central queue, multiple queues with affinity mapping, multiple queues with work stealing, and variants of these algorithms with priorities. However, FLAME does not implement task and data transfer cost modelling, and no support is currently provided for programming clusters.

The ParalleX project [17] introduces a new, innovative parallel and distributed programming model for scaling applications on peta-scale and larger computing clusters. This new model is built on the concept of message-driven work-queues with an active global address space for simplifying load balancing work. The ParalleX model is implemented by the HPX runtime [18]. A concept called *percolation* is mentioned in [19] as being dedicated to accelerators, without any further details.

The new implementation of the Adaptive Partitioning Controller APC+ [20] provides a distributed component-based dataflow system with load balancing support through a work stealing mechanism and an optimized communication support for aggregating data transfers. APC+ addresses heterogeneity through cost modeling. However, APC+ only targets applications with a n-dimensional workspace finitely divisible in work-tiles. Moreover, APC+ assumes a uniform processing time per work-tile size, which is an issue for applications built with multiple computing kernels.

The Qilin [21] approach for adaptively mapping computations on a single CPU/GPU node is based on building a DAG and optimizing its mapping on the computing node through a dynamic compilation step at the program start. The optimizing step uses a database of past execution times of the program to decide the mapping of work on processing elements. However, this approach suffers from the same issue as APC+ in that it optimizes for a single computation kernel, and Qilin only targets single nodes while APC+ targets distributed platforms.

The DAGuE [22] support for micro-task management (recently renamed PaRSEC) implements a distributed scheduler. Its specificity is to perform scheduling directly on a concise, algebraic DAG representation, without ever unrolling the DAG into memory. This approach offers a significant advantage in terms of scalability on clusters. However, as mentioned in [22], this approach does not work for applications whose DAG is dynamically computed throughout the application, while StarPU can schedule such dynamically built DAGs. Related past works at UTK's ICL also include TBlas [23, 24], a linear algebra library for distributed multicore systems with predefined block distribution and a lightweight scheduler, and the Quark [25] runtime for shared-memory multicore systems.

The message-driven object runtime system Charm++ supports heterogeneous clusters [26–28] as an extension to its uniform *chare object*-based programming model on top of C++. The entry methods of the chare objects are extended with *accelerated* entry methods to indicate work to be potentially executed on accelerators. At the runtime level, Charm++ implements an Accelerator Manager to coordinate the execution of accelerated entry methods using performance metrics monitoring together with load balancing capabilities. Thus, unlike StarPU, the Charm++ runtime works in a *reactive way* upon workload imbalance among heterogeneous processing elements detected through busy/idle time monitoring as well as periodic measurement.

The fundamental approach of the XKaapi runtime is to implement task scheduling and load balancing through work stealing. The XKaapi runtime has been extended to support scheduling on a heterogeneous multi-GPU node [29,

30]. This extension implements an *affinity list* to favor gathering interacting objects on the same processing unit, as well as a dynamic threshold mechanism to decide whether some work is worth the cost to be executed on a GPU or should be executed on CPU. XKaapi does not provide cluster support however.

ETI SWARM [31] (SWift Adaptive Runtime Machine) is a commercial codelet-based runtime dedicated to scalability. It uses component groupings structured as a tree on a distributed platform. Very few details are given about the scheduling strategies at the cluster level and within heterogeneous nodes.

6 Conclusion

We have shown how the StarPU task paradigm can be combined with MPI communications, and then how tasks can be used as first-class citizen by mapping DAGs of tasks and the corresponding data on the different MPI nodes. Even though modifying large MPI codes remains a real concern, StarPU can be used within parallel libraries or higher-level programming environments using a task-based paradigm internally. This is available in the stable version of StarPU and currently being integrated in the MAGMA reference linear algebra library to extend it to distributed systems.

Our work can still be improved on several aspects. Latency and bus contention being major concerns on a cluster, we will support specific capabilities such as direct transfers between GPUs and NICs, and having a better integration of low-level toolkits within MPI would be useful. Besides, we would need to improve StarPU's scheduling engine to consider network activity. We could for example model the performance of the network to predict when data should be available on the different MPI nodes.

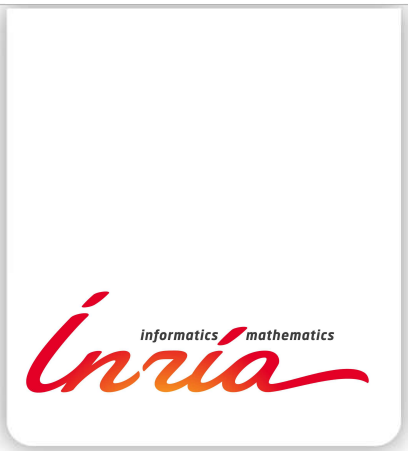
In order to cope with the asynchronous nature of task parallelism, the MPI standard needs to evolve to fit applications that are not written in a synchronous SPMD style. For example, we would need asynchronous collective operations (*e.g.* a non-blocking scatter/gather MPI call) to efficiently dispatch the output of a task to all MPI nodes which may all need it at a different time. The thread safety of MPI implementations are also still a concern, and we are looking forward good thread-safe MPI implementations to be able to submit data requests any time without having to use a dedicated thread.

When exploiting very large MPI clusters, registering all data, and having all nodes run through the whole DAG becomes inefficient. The application can actually already avoid registering data and submitting tasks on nodes which will not interact with them. It would however be useful to build an intermediate layer which does this optimization automatically. Dynamic distribution of tasks over MPI nodes would permit to avoid requiring a static distribution of data. Dynamic load balancing over a cluster is however a delicate issue. Similarly to out-of-core algorithms, we need for instance to ensure that the memory footprint of the application is not too high when migrating on the same MPI node tasks which access many pieces of data.

References

1. Jenkins, J., Balaji, P., Dinan, J., Samatova, N.F., Thakur, R.: Enabling Fast, Non contiguous GPU Data Movement in Hybrid MPI+GPU Environments (2012)
2. Technologies, M.: NVIDIA GPUDirect Technology Accelerating GPU-based Systems. `http://www.mellanox.com/pdf/whitepapers/TB_GPU_Direct.pdf` (2010)
3. Augonnet, C., Clet-Ortega, J., Thibault, S., Namyst, R.: Data-Aware Task Scheduling on Multi-Accelerator based Platforms. In: The 16th International Conference on Parallel and Distributed Systems (ICPADS). (2010)
4. Agullo, E., Augonnet, C., Dongarra, J., Ltaief, H., Namyst, R., Thibault, S., Tomov, S.: A Hybridization Methodology for High-Performance Linear Algebra Software for GPUs. In Wen-mei W. Hwu, ed.: GPU Computing Gems. (2010)
5. Agullo, E., Augonnet, C., Dongarra, J., Ltaief, H., Namyst, R., Roman, J., Thibault, S., Tomov, S.: Dynamically scheduled Cholesky factorization on multicore architectures with GPU accelerators. In: Symposium on Application Accelerators in High Performance Computing (SAAHPC), Knoxville, USA (07 2010)
6. Agullo, E., Augonnet, C., Dongarra, J., Faverge, M., Langou, J., Ltaief, H., Tomov, S.: LU factorization for accelerator-based systems. In: The 9th ACS/IEEE International Conference on Computer Systems and Applications. (2011)
7. Agullo, E., Augonnet, C., Dongarra, J., Faverge, M., Ltaief, H., Thibault, S., Tomov, S.: QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators. In: 25th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2011). (5 2011)
8. Blackford, L.S., Choi, J., Cleary, A., D'Azeuedo, E., Demmel, J., Dhillon, I., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., Whaley, R.C.: ScaLAPACK user's guide. Society for Industrial and Applied Mathematics (1997)
9. Anderson, E., Bai, Z., Dongarra, J., Greenbaum, A., McKenney, A., Du Croz, J., Hammerling, S., Demmel, J., Bischof, C., Sorensen, D.: Lapack: a portable linear algebra library for high-performance computers. In: The 1990 ACM/IEEE conference on Supercomputing
10. Chen, J., III, W.W., Mao, W.: GMH: A Message Passing Toolkit for GPU Clusters. In: 16th International Conference on Parallel and Distributed Systems. (2010)
11. Barak, A., Ben-Nun, T., Levy, E., Shiloh, A.: A Package for OpenCL Based Heterogeneous Computing on Clusters with Many GPU Devices. In: Workshop on Parallel Programming and Applications on Accelerator Clusters (PPAAC). (2010)
12. Zheng, Y., Iancu, C., Hargrove, P.H., Min, S.J., Yelick, K.: Extending Unified Parallel C for GPU Computing . In: SIAM Conference on Parallel Processing for Scientific Computing (SIAMPP). (2010)
13. Lee, J., Tran, M.T., Odajima, T., Boku, T., Sato, M.: An Extension of XcalableMP PGAS Lanaguage for Multi-node GPU Clusters . In: HeteroPar. (2011)
14. Bueno, J., Planas, J., Duran, A., Martorell, X., Ayguad, E., Badia, R.M., Labarta, J.: Productive Programming of GPU Clusters with OmpSs . In: Parallel and Distributed Processing Symposium (IPDPS). (2012)
15. Igual, F.D., Chan, E., Quintana-Ort, E.S., Quintana-Ort, G., van de Geijn, R.A., Zee, F.G.V.: The FLAME approach: From dense linear algebra algorithms to high-performance multi-accelerator implementations. Journal of Parallel and Distributed Computing (2012)
16. Quintana-Ort, G., Igual, F.D., Quintana-Ort, E.S., van de Geijn, R.A.: Solving dense linear systems on platforms with multiple hardware accelerators. In: Symposium on Principles and Practice of Parallel Programming, PPoPP. (2009)

17. Heller, T., Kaiser, H., Iglberger, K.: Application of the parallex execution model to stencil-based problems. In: International Supercomputing Conference. (2012)
18. Kaiser, H., Brodowicz, M., Sterling, T.: Parallex: An advanced parallel execution model for scaling-impaired applications. In: Parallel Processing Workshops. (2009)
19. Tabbal, A., Anderson, M., Brodowicz, M., Kaiser, H., Sterling, T.: Preliminary design examination of the parallex system from a software and hardware perspective. SIGMETRICS Performance Evaluation Review (2011)
20. Hartley, T.D.R., Saule, E., mit V. atalyrek: Improving performance of adaptive component-based dataflow middleware. Parallel Computing (2012)
21. Luk, C.K., Hong, S., Kim, H.: Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: The 42nd Annual IEEE/ACM International Symposium on Microarchitecture. (2009)
22. Bosilca, G., Bouteiller, A., Danalis, A., Hrault, T., Lemarinier, P., Dongarra, J.: DAGuE: A generic distributed dag engine for high performance computing. Parallel Computing (2012)
23. Song, F., Ltaief, H., Hatem, B., Dongarra, J.: Scalable tile communication-avoiding qr factorization on multicore cluster systems. In: The Conference for High Performance Computing, Networking, Storage and Analysis. (2010)
24. Song, F., YarKhan, A., Dongarra, J.: Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In: The Conference on High Performance Computing Networking, Storage and Analysis. (2009)
25. YarKhan, A., Kurzak, J., Dongarra, J.: QUARK Users' Guide: QUeueing And Runtime for Kernels. UTK ICL. (2011)
26. Kunzman, D.: Runtime support for object-based message-driven parallel applications on heterogeneous clusters. PhD thesis, University of Illinois (2012)
27. Kunzman, D.M., Kal, L.V.: Programming heterogeneous clusters with accelerators using object-based programming. Scientific Programming (2011)
28. Zheng, G., Meneses, E., Bhatele, A., Kale, L.V.: Hierarchical load balancing for charm++ applications on large supercomputers. In: P2S2. (2010)
29. Gautier, T., Lima, J.V.F., Maillard, N., Raffin, B.: Locality-aware work stealing on multi-cpu and multi-gpu architectures. In: Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG). (2013)
30. Hermann, E., Raffin, B., Faure, F., Gautier, T., Allard, J.: Multi-gpu and multi-cpu parallelization for interactive physics simulations. In: The 16th international Euro-Par conference on Parallel processing: Part II. (2010)
31. Lauderdale, C., Khan, R.: Towards a codelet-based runtime for exascale computing: position paper. In: The 2nd International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era. (2012)



RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399